Implementing Unix Signals

Tera Computer Company
400 N. 34th St.
Seattle, WA 98103

April 6, 1993

1 Introduction

This document discusses how the Tera OS implements Unix and Posix signals. It introduces Unix signal concepts, describes the type of signals used by Posix, Berkeley Unix and AT&T V.4 Unix. It then describes the Tera OS implementation of signals.

2 Unix Signals

Unix signals are a version of software interrupts. They cover two distinct types of events - asynchronous signals and exceptions. Aysnchronous signals are asynchronous events delivered from external agents (e.g. a time out or a Control-C from typed from the keyboard). Exceptions are synchronous events resulting from internal actions (e.g. a segmentation violation or division by zero.).

A signal that has been sent is called `pending`. Normally the interval between sending a signal dispatch and its delivery is not detectable by the receiving task. However, a task can specify a set of signals to be `blocked`, i.e. signals to remain pending until they are specifically unblocked by the task. Tasks can query about the state of pending signals.

When a signal arrives, Unix will complete system calls in progress except for those that can take a long time. These calls include: `read()`/`write()` to a "slow" device (e.g. a terminal but not a file), `fcntl()`, `ioctl()`, `wait()`, and `waitid()`. Processes sleeping uninterruptably (e.g. in disk wait) do not have signals delivered. Unix defers signal delivery until the event the process slept on has completed.

Upon receiving a signal, a task will do one of three things:

1. if the task is ignoring this signal, it is discarded.
2. if a signal handler has been installed for this signal, that code is executed.
3. otherwise, the default action associated with this signal is performed.

All signals have an associated default action. Possible defaults include:

- ignoring the signal

- terminating the task
- terminating the task after a *core* file has been generated
- suspending the task.

Some signals can not be ignored, blocked or handled by the user (e.g. an unconditional termination signal).

The AT&T Unix implementation of signals through System V.3 did not provide reliable signals, i.e. race conditions existed. Berkeley 4.4 (4.4BSD), Posix and AT&T System V.4 signals are reliably delivered, can be ignored or blocked, and blocked signals remain pending until unblocked. The Tera OS implements reliable signals, the exact flavor of which (Posix, System V.4, etc.) remains to be decided.

System V.4 and Berkeley 4.4 provide different signal implementations while Posix defines its standard as something in between. The next sections describe the various implementations and their differences.

A good description of Unix signals can be found in [Bac86] and [LMKQ89].

3 Posix Signals and System Call Interface

A complete description of the Posix signal standard can be found in [Pos88]. The following summary came from that source.

3.1 Signals

Posix requires the following signals:

- SIGABRT - Abnormal termination signal, such as is initiated by the `abort`() function.
- SIGHUP - Hangup detected on controlling terminal or death of controlling process.
- SIGALRM - Timeout signal, such as initiated by the `alarm`() function.
- SIGFPE - Erroneous arithmetic operation, such as division by zero or an operation resulting in overflow.
- SIGINT - Interactive attention signal.
- SIGKILL - Termination signal (cannot be caught or ignored).
- SIGPIPE - Write on a pipe with no readers.
- SIGQUIT - Interactive termination signal.
- SIGILL - Detection of an invalid hardware instruction.
- SIGSEGV - Detection of illegal memory reference.
- SIGTERM - Termination signal.

- SIGUSR1 - Reserved as application-defined signal 1.
- SIGUSR2 - Reserved as application-defined signal 2.

Posix also defines the following Job Control signals for systems that implement BSD style job control:

- SIGCHLD - Child process terminated or stopped.
- SIGCONT - Continue if stopped.
- SIGSTOP - Stop signal (cannot be caught or ignored).
- SIGSTP - Interactive stop signal.
- SIGTTIN - Read from control terminal attempted by a member of background process group.
- SIGTTOU - Write to control terminal attempted by a member of a background process group.

3.2 System Interface

3.2.1 Signal Sets

A signal set is an application defined data object consisting of one or more signals. Posix provides a set of primitives to operate on signal sets.

```
int sigemptyset(sigset_t* set); // Initializes set to empty.
int sigfillset(sigset_t* set) // Initializes set to full.
int sigaddset(sigset_t* set, int signo) // Add signo to set.
int sigdelset(sigset_t* set, int signo) // Delete signo from set.
int sigismember(sigset_t* set, int signo) // Returns 1 if signo is a member
                                          // of set, otherwise 0.
```

3.2.2 Binding Signal Actions

To examine or change signal actions, Posix provides

```
int sigaction(int sig, struct sigaction* act, struct sigaction* oact);
```

which allows the calling process to examine or specify (or both) the action to be associated with a specific signal. The sigaction structure contains a pointer to a handler, a mask and a set of flags. The handler consists of SIG_DFL (default action), SIG_IGN (ignore the signal), or a pointer to a user specified handler. The mask contains an additional set of signals to be blocked during the execution of the signal-catching function. The flags are used to modify the behavior of the specified signal (read "hack").

3.2.3 Blocking, Pending, etc.

The set of signals being blocked can be changed or examined using:

int sigprocmask(**int** how, sigset_t* set, sigset_t* oset).

How has one of three values:

- SIG_BLOCK - Block the union of currently blocked signals and *set.*
- SIG_UNBLOCK - Unblock the intersection of currently blocked signals and *set.*
- SIG_SETMASK - Replace the current set of blocked signals with *set.*

To examine which signals are pending:

int sigpending(sigset_t* set)

which stores a set of signals that are blocked from delivery and pending arrival. To wait for the arrival of a signal:

int sigsuspend(sigset_t* sigmask);

The process relinquishes the processor with its process signal mask set to *sigmask.* To send a signal one uses

int kill (pid, sig);

which sends signal *sig* to process *pid.*

4 Berkeley 4.4 Signals

Berkeley 4.3BSD [LMKQ89] implements Posix signals, except for SIGABRT and different system call names. Berkeley 4.4BSD provides for SIGABRT and all the Posix interfaces. Berkeley also provides following additional signals:

- SIGTRAP - Trace trap.
- SIGIOT - I/O trap instruction executed.
- SIGEMT - Emulate instruction executed.
- SIGBUS - Bus Error.
- SIGSEGV - Segmentation violation.
- SIGSYS - Bad argument to system call.
- SIGURG - Urgent condition on I/O channel.

- SIGIO - I/O possible on a descriptor.
- SIGXCPU - CPU time limit exceeded.
- SIGXFSZ - File size limit exceeded.
- SIGVTARM - Virtual timer exceeded.
- SIGPROF - Profiling timer exceeded.
- SIGWINCH - Window size changed.

BSD signals also provide an alternate stack mechanism that allows a signal hander to use a different stack. This is important for handling stack overflows and similiar situations.

4.1 System Call Interface

The 4.3BSD system call interface provided the basis for the Posix standard so that the differences are generally minor. The major difference is that Posix does not provide an alternative stack mechanism. Table 4.1 provides a comparison of the two interfaces.

4.1.1 Alternate Stack

Normally a signal handler uses the current stack while executing. The sigstack() call allows the user to provide an alternate stack. This ability can be vital if the reason the signal occurred was due to a stack overflow.

5 System V.4 Signals

With System V.4, AT&T adopted much of the Posix and Berkeley signal apparatus. They have added job control and reliable signals. Much more detail can be found in the following System V.4 manuals: [Sys90b], [Sys90c], and [Sys90a].

4.3BSD	Posix
sigmask()	sigsetops()
sigblock()	sigprocmask()
sigsetmask()	sigprocmask()
sigpause()	sigsuspend()
sigvec()	sigaction ()
sigstack()	none
sigreturn()	none
none	sigpending()

TABLE 1: Comparing 4.4BSD and Posix System Call Interface

Signals added to the Posix standard that come from 4.3BSD are: SIGILL, SIGTRAP, SIGEMT, SIGBUS, SIGSYS, SIGPWR, SIGWINCH, SIGURG, SIGVTARM, SIGPROF, SIGXCPU, SIGXFSZ, SIGIOT, and SIGIO. The signals added for implementing job control are the same as in the Posix section.

System V.4 has added numerous flags option to **sigaction**(). Many seem dedicated to providing System V.3 unreliable signals.

- SA_ONSTACK - Use an alternate stack (See Section 4.1.1).
- SA_RESETHAND - Reset signal disposition to SIG_DFL when signal is caught.
- SA_NODEFER - Do not block this signal while it is being handled.
- SA_RESTART - A system call interrupted by a caught signal is automatically restarted.
- SA_SIGINFO - Pass handler, in addition to sig, a pointer to a struct siginfo_t and a struct u_context.
- SA_NOCLDWAIT - If sig SIGCHLD, don't create zombies for exiting children.
- SA_NOCLDSTOP - If sig SIGCHLD, don't signal process when children stop or continue.

5.1 System Interface

The system interface follows the Posix interface but differs in some important ways. An outline of the interface is given below, complete details are in [Sys90c] .

5.2 sigaction

int sigaction (**int** sig, **const struct** sigaction *act, **struct** sigaction *oact);

This call permits the caller to examine or change the actions associated with a signal. The `sigaction` structure includes a handler, a mask of signals to be blocked while in the handler, and a set of flags (see 5).

5.3 sigaltstack

int sigaltstack (**const** stack_t *ss, stack_t *oss);

`sigaltstack` allows a process to provide an alternate stack for use during signal to be used while processing signals. This is essential when handling stack overflow signals and can be useful in other situations. The `stack_t` structure holds the location of the alternate stack, its size and flags affecting the usage of the call.

5.4 Simplified Signal Management

```
void (*signal (int sig, void (*disp) (int))) (int);
void (*sigset (int sig, void (*disp) (int))) (int);
int sighold(int sig);
int sigrelse(int sig);
inst sigignore(int sig);
int sigpause(int sig);
```

These functions provide a subset of the capabilities of `sigaction` but can be used for simple signal management. For those without super C parsing skills, `signal` is a function that takes two arguments: an integer *sig* and a pointer to a function *disp*. *disp* takes an integer as a parameter and returns void. `signal` returns a pointer to a function returning void and that takes an integer parameter. Basically, `signal` returns a pointer to something like *disp*.

`signal` sets the signal handler of *sig* and returns the old handler. `sigset` is similar but adds *sig* to the signals blocked while in the handler (providing reliable signals). `sighold` adds the signal to the process's signal mask while `sigrelse` removes it. `sigignore` adds the signal to the set of ignored signals. `sigpause` removes the signal from those signals being blocked and suspends the process until a signal is received.

5.5 sigsend

```
int sigsend(idtype_t idtype, id_t id, int sig);
int sigsendset(procset_t *psp, int sig);
```

These calls are generalized mechanisms for sending a signal to a process or a set of processes. *idtype* specifies if *id* is a process, process group, session, user or group ID. *psp* is a binary tree structure that allows all sorts of complicated stuff; e.g. apply operation to exclusive-or of the two sides of the tree. I'm sure someone at AT&T decided they **really** needed this because its not clear who or what would ever use it.

5.6 Posix Style Calls

The calls `sigprocmask`, `sigpending`, `sigsuspend` appear the same as the POSIX calls - see the Posix section for more information or the [Sys90c].

5.7 Subroutine Level Calls

System V.4 provides the same signal set calls as Posix. See section 3.2.1.

6 Tera Implementation of Unix Signals

The Tera OS attempts to provide a POSIX interface to UNIX signals. Since our programming model differs from the standard single threaded Unix process, the mapping won't be exact. However, the OS works hard to present a signal implementation that allows standard Unix programs to work "as is" with few exceptions. To this end, user level signal handling is done in a serial environment.

The Tera OS handles synchronous and asynchronous signals very differently. Synchronous signals (also known as exceptions) are generated internally by the execution of the code. Examples include arithmetic errors and address violations. On the Tera, these generate traps which are handled by the user run time (URT) trap handler **NOT** the OS. If the URT can not successfully handle the exception, only then is it passed to the OS (via a system call, e.g. `kill()` or `vm_fault()`). The paper [?] explains how the URT handles exceptions. For the rest of this document we will focus exclusively on asynchronous signals. This includes: time outs; keyboard interrupts; and system call generated signals(e.g. `kill()`).

The OS provides many system calls to the URT for use in signal handling. A complete description of the interface is found in [?]

6.1 Signal Data Structures

Unix keeps all of the relevant signal information in fields in the `proc` structure. Since the Tera OS maintains a `proc` entry for each task, the signal fields in `proc` remain the same. Most signal activity takes place in the supervisor layer except for a portion of the actual signal delivery that occurs in the kernel. This is supported using a few signal specific fields in kernel data structures.

6.2 Implementing Signals Using the BSD Kernel

The heart of the BSD Unix signal handling is in the file `kern_sig.c`. The original intention was to make as few changes as possible to the original code so as to preserve the original Unix semantics. As the implementation proceeded, it became clear that this would not be possible. As a result, most of the code to post and deliver signals was completely rewritten. The routines: `issig`, `stop`, `psig`, and `sigexit` were removed. The signal post routine `psignal` was complete rewritten. New routines were added:

- stopProcess: replaced `stop`. Stops the process, notifies the parent and marks the process as stopped.
- continueProcess: Resumes the process and marks it as not stopped.
- doStopContSigs: Determine if a the process has a signal that stops or continues it. Call one above routines to do the action.
- deliverSignal: This routine is executed by a supverisor daemon. It checks that the process has signals to deliver and takes the appropriate action.
- killProcess: Forces the process to exit with the specified signal, dumping core if appropriate.
- checkFatalSigs: This routine is called by `tera_sched_swapsave_complete()` in the supervisor code. It checks for signals that terminate the process. If any are found, it returns TRUE unless 'die' is TRUE in which case it creates a coredump if needed and calls `killProcess()`.
- pendingSigs: This routine is called by `tera_sched_swapsave_complete()` in the supervisor code. This routine returns 1 if there are pending signals not being held else 0
- caughtSigs: This routine returns the next pending signal that is caught or -1 if none are found.

6.3 Signal System and Libary Calls

This section describes the implementation of each POSIX required call.

6.3.1 sigaction - Setting a Signal Handler

The `sigaction()` call sets the action associated with a signal handler. On the Tera, `sigaction()` is a user library routine that allows the URT to catch the call and register the handlers. The URT informs the OS of the signal action (i.e. whether the signal is handled, ignored, etc) using the system call `tera_sigaction()`, but the handler information is not used.

6.3.2 kill - Sending a Signal

The `kill` system call sends a signal to a given process or group of processes. Section 6.4 contains the details of sending signals. The implementation does not change the existing Unix `kill` code.

6.3.3 sigsetops - Manipulating Signal Sets

The routines: `sigemptyset`, `sigfillset`, `sigaddset`, `sigdelset`, and `sigismember`, manipulate `sigset_t` structures. Since these are user addressable structures, all the routines are library routines in libc and are used as is.

6.3.4 sigprocmask - Examine and Change Blocked Signals

This system call examines and/or changes which signals the signal mask blocks. If the sigprocmask operation results in signals being deliverable, the SPChore does a tsleep() to block until at least one signal is delivered. This follows Unix semantics. This remains a system call under Tera OS and uses existing Unix kernel code.

6.3.5 sigpending - Examine Pending Signals

This system call returns a `sigset_t` of which signals are pending. This remains a system call under the Tera OS and uses existing Unix kernel code.

6.3.6 sigsuspend - Suspend and Wait For a Signal

This system call blocks the process under Unix. There are two possible semantic interpretations of this call. One way is to swap the task out regardless of whether other stream activity is occurring. The other allows the task to continue. The first cut will be to implement the second style.

On the Tera, `sigsuspend` will be a blocking system call that does a upcall to the URT. If the process does not have any other activity, the program blocks using sched blocking mechanisms already in place. The `sigsuspend` call returns once a signal has been delivered.

6.3.7 alarm - Scheduler an Alarm

This system call schedules an event which results in delivery of a `SIGALRM` at the specified time. Under Tera this call becomes a library routine that calls `setitimer` for the real time timer.

`setitimer` and `getitimer` remain system calls that use the existing Unix kernel code. The code uses the Unix `timeout` facility which Tera reimpliments in C.

6.3.8 pause - Suspend

This call is similar `sigsuspend` using the current signal mask.

6.4 Sending an External Signal

For a user stream to send a signal to another task or itself, it makes the `kill` system call. The supervisor promoted chore (SPChore) has its permissions and parameters verified by Unix code, then executes Unix kernel `kill`() code. `kill`() (or `killpg`() if sending to a process group) checks to see if the caller can send a signal to that process or process group and posts the signal via `psignal`().

The code in `psignal`() checks a large variety of special cases but in general it looks like this (in pseudocode):

```
get signal action
if action is |IGNORE| then return
if action is |DEFAULT| and default action is |IGNORE| then return
if action is |HOLD|, then mark it pending and return
if action is |DEFAULT| or |HANDLED| then
    mark task as having signal pending
    if task does not have uninterruptable sleeping chores
        create a supvDaemon to call deliverSignal
endif
```

The signal will be delivered either when a sched starts to run or when the supervisor daemon executes `deliverSignal`() for that task.

6.5 Receiving an External Signal

In Unix, signals are delivered when the user returns from the kernel. This happens when returning to user space from a system call or when the kernel has preempted the user to process a clock interrupt.

On the Tera, if task was not executing when the signal was posted, the OS delivers the signal during an mp-swap. The URT asks for pending signals before restoring its state. This approximates the Unix methodology.

If the task is executing, there's a problem with timely delivery of signals. The Tera OS doesn't preempt the user to service clock interrupts, the user might not make system calls, and the task

might not pm-swap soon (or ever) so the OS needs a different stategy to deliver the signals to a running task.

The OS delivers the signal by having the URT save its state and then request pending signals as if the task was just mp-swapped. This method ensures serial execution of signal handlers, timely signal delivery, and uses existing mechanisms. Note that this method is for handled signals. If the user does not cooperate with the OS by following required guidelines, the only user handled signals are affected.

Some thread of control needs to deliver the signal. The OS uses the supervisor daemon facility to spawn a supervisor daemon which restarts the task using a "fake" pmSwap. A "fake" pmSwap entails having the URT save its state as in a real pm-swap but the OS never actually swaps it out of the protection domain(s). This saves the overhead of unloading and reloading protection domain state (e.g. address maps). After the restart, the sched asks for any pending signals.

Here are the steps that occur for restarting a running task:

1. The supervisor daemon executes the unix routine `deliverSignal()`.
2. `deliverSignal()` checks that signals are pending and executes any pending stop or resume signals. If the task is suspended and there are fatal signals or if there are fatal/caught signals, it restarts the task using the klib routine `restartTask()`.
3. `restartTask` initially acts like `pmSwap()` by setting the domain signal for each team and waiting for a reply message.
4. The URT saves its state believing a pm-swap is in progress. Then it calls `team_swapsave_complete()` which sends a message to the stream waiting above. If it's not the last team in the sched, the SPChore spins on `SPTask :: signalBarrier`. Otherwise it returns a non-zero value to the URT.
5. When stream in `TeamControlBlock_k :: postSignal` receives the message, it returns rather than actually unloading the team as in a pm-swap. [Note: the reason it waits around for a message is that if the URT never complies with the domain signal, then the task must be killed off]
6. When the URT gets a non-zero return from `team_swapsave_complete()` it saves sched counters and calls `sched_swapsave_complete()`.
7. The SPChore verifies that it is the only stream in the protection domain. Then it calls `schedSwapped` klib routine which notes that this sched is swapped.

If the sched had done a real pm-swap and was now mp-swapping, the pb-scheduler restarted the teams via `TeamControlBlock :: mp-swap()`. These streams re-enter `team_swapsave_complete()` (except for the one that did the `sched_swapsave_complete()` which re-enters that) and spin on the `SPTask :: signalBarrier`.

At this point, the sched looks very similar to a mp-swapping sched and delivery of signals is the same.

1. In `sched_swapsave_complete()`, if the sched had been mp-swapped then the SPChore returns from `SPTeam :: haltSwapChore()`. Otherwise the SPChore skips the `SPTeam :: haltSwapChore()` call. The SPChore checks if a signal is being delivered, see that it is, and calls `SPSched_k :: wakeSleepingChores()`.

2. SPSched_k :: wakeSleepingChores() runs through the list of blocked SPChores for any non-interruptable sleepers. If any are found, the signal delivery is aborted since under Unix semantics, processes sleeping uninterruptably don't have signals delivered. Otherwise, it restarts all blocked (sleeping) SPChores. Each will eventually return from the system call and put itself on the SPTeam Unix_Return list.

3. Once all sleepers have finished, the SPTask barrier is lowered and the any spinning streams return to the URT.

4. In the URT, one stream per sched determine it is the master while the others spin. The master gets a new ccb and queries the OS with the signal_number() system call.

5. In signal_number(), the SPChore verifies that things are as they should be (no other streams running, mp-swap in progress, etc.). It then does the CURSIG() macro to get the first signal and calls psig() to do it.

6. In psig():

   - If the signal causes an exit or coredump, it calls sigexit () which calls taskTerminate().
   - If the signal is handled: return the signal number which is returned to the URT. [Unix calls sendsig() which plays around with the user's return stack so that when the user returns it executes the signal handler.]

7. The URT finds the signal handler and executes it.

8. If the user handler executes a longjmp(), the URT finds the chore that made the setjmp() call and unwinds it. The details of how the URT handles this case can be found in [?]. The URT then makes a did_longjmp() system call with two parameters. The first is a signal mask associated with the setjmp buffer that reestablishes that signal mask or NULL if the mask remains unchanged. The second is used to abort a system call if the chore that did the setjmp() was blocked in a system call. If it was blocked the URT gives the unique id associated with the upcall, otherwise it gives it a NULL.

9. If the handler returns, the URT calls signal_number() again.

10. The OS gets the value from psig() until psig() returns -1 meaning no more signals are pending. It then reenables SPChore restarts and returns -1 to the URT.

11. At this point that signal handling is completed, the URT restarts the rest of the scheds.

7 How Tera Signal and Unix Signals Differ

Tera is using BSD4.4 as its code base. Since Posix signals are based on BSD signals the Tera implementation provides a very close rendition of Posix signals. As of this time, no known inconsistencies exist. This situation is subject to change as the implementation occurs. The only known limitation for the signal implementation is that longjmp out of signal handlers is not supported for multi-sched tasks.

Posix has a draft document detailing how signals interact with Posix threads. The draft specifies two types of implementations: one with signals per process and the other per thread. Since signals

on the Tera are on a per process/task basis, we follow the former. This draft document provides a few additional signal calls for threads. Since Tera is not currently providing a Posix Threads package, none of the additional calls are implemented.

BSD and SysVR4 signals allow alternate signal stacks. The Tera implementation does not support those calls directly. However, the URT will probably provide a new CCB for the chore executing the normal situation will be to use an alternate stack.

SysVR4 provides numerous flags to its calls that Tera OS does not currently support.

References

[Bac86] Maurice J. Bach. *The Design of the Unix Operating Systems.* Prentice-Hall, Inc., 1986.

[LMKQ89] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD Unix Operating System.* Addison-Wesley Publishing Company, 1989.

[Pos88] *IEEE Standard Portable Operating System Interface for Computer Environments - IEEE Std 1003.1-1988.* The Institute of Electrical and Electronics Engineers, Inc., 1988.

[Sys90a] *Unix System V Release 4 BSD/XENIX Compatibility Guide*, 1990.

[Sys90b] *Unix System V Release 4 Programmer's Guide: POSIX Conformance*, 1990.

[Sys90c] *Unix System V Release 4 Programmer's Reference Manual*, 1990.